A High-Performance Distributed Object-Store for Exascale Numerical Weather Prediction and Climate

Simon D. Smart
European Centre for Medium Range Weather Forecasts
Reading, United Kingdom
simon.smart@ecmwf.int

Tiago Quintino
European Centre for Medium Range Weather Forecasts
Reading, United Kingdom
tiago.quintino@ecmwf.int

Baudouin Raoult
European Centre for Medium Range Weather Forecasts
Reading, United Kingdom
baudouin.raoult@ecmwf.int

ABSTRACT

Numerical Weather Prediction (NWP) and Climate simulations sit at the intersection between classically understood High Performance Computing (HPC) and Big Data / High Performance Data Analytics (HPDA). Driven by ever more ambitious scientific goals, both the size and number of output data-elements generated as part of NWP operations have grown by several orders of magnitude, and are expected to continue growing exponentially in the future. Over the last 30 years this increase has been approximately 40% per year. To cope with this projected growth, ECMWF has been actively exploring novel hardware and software approaches to workflow and data management. ECMWF's meteorological object store acts as a hot-cache for meteorological objects within the forecast pipeline, and supports multiple backends to enable the use of different storage technologies. This paper presents extensions to this object store to allow it to operate in a distributed fashion on a wider range of hardware without assuming the presence of high-performance, parallel, globally namespaced storage systems. The improvements include a flexible, configurable front end which gives control of where data is to be stored without requiring code changes in the calling application.

CCS CONCEPTS

• **Information systems → Indexed file organization**; *Hierarchical data models*; *Hierarchical storage management*; *Environment-specific retrieval*; B-trees; Version management.

ACM Reference Format:
Simon D. Smart, Tiago Quintino, and Baudouin Raoult. 2019. A High-Performance Distributed Object-Store for Exascale Numerical Weather Prediction and Climate. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '19), June 12–14, 2019, Zurich, Switzerland.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3324989.3325726

1 INTRODUCTION AND MOTIVATION

The European Centre for Medium-Range Weather Forecasts (ECMWF) is an international organisation whose principle objectives are the development and operational delivery of numerical weather forecasts, and the archival and distribution of meteorological data. Forecasts are produced in four one-hour time critical windows per day. Meteorological output data is generated as a stack of two-dimensional slices of the atmosphere surrounding the globe like layers of an onion. These slices are known as fields and their size grows quadratically as spacial resolution increases. In bursts of one hour time-critical windows, the forecast system currently generates around 130 TiB of output per day (22M fields).

The data is stored in byte streams described by GRIB format [33], which is standardised by the World Meteorological Organisation for compatibility between meteorological centres. This format is intricate, requiring specialised tools to decode and interpret, but self-describing, such that metadata can be extracted from a field. This metadata takes the form of a structured set of key-value pairs. Globally, the available metadata space is extremely sparsely occupied, but with very dense regions.

The Fields Database (FDB) is a software library, and an internally provided service, used as part of the weather forecasting software stack. It operates as a domain-specific object store for byte streams of meteorological data, such that the output from ECMWF's Integrated Forecasting System (IFS) is written into the FDB, from where it is retrieved by the various post-processing and archival tasks. In this capacity, the FDB also operates as the highest layer within an application controlled hierarchical storage manager.

The Meteorological Archival and Retrieval System (MARS) [21] is a primary service offered by ECMWF that makes many decades of meteorological observations and forecasts available to a wide range of end users and operational systems. At the base of the stack is the tape archive built on the IBM High Performance Storage System (HPSS) [32] which is supported by the MARS disk-based cache. The FDB sits between the HPC systems and the rest of the MARS infrastructure, absorbing the forecast output and making it available throughout the post-processing pipeline and elsewhere, efficiently working as a first line cache within the HPC.

In practice, operational weather forecasts decay in value very rapidly after they are made (being superseded by forecasts made on later occasions). The FDB exists to make this data quickly and cheaply available while it is broadly useful and thus accessed frequently. Operational data in the FDB has a lifetime of between 3 and 5 days. Because the data flows through this system are predictable, they are application controlled and are thus optimised for performance.

In current operations, roughly 200 TiB of data pass through the FDB per day (including both operations and research activity). More than 100 TiB of this data is then moved to MARS for archival. At any given time, the total contents of the operational FDB are estimated to be between 4 and 5 PiB.

The MARS infrastructure uses the key-value pair nature of the metadata to structure its operational language. Data write requests use the complete metadata associated with a field to index it. Data accesses use either complete or partial sets of key-value pairs to navigate the indexing information and access the data objects. Which keys are required is well defined according to a schema that guides the data collocation policy.

There are a number of challenges on the horizon. The amount of data being generated and processed is not only large, it is growing rapidly. In 1995, operations generated a total of 14 TiB per year, whereas by the end of 2018 just the forecast model was generating 20 TiB in one hour. This is currently made up of 7M fields, each between 1 and 20MiB. To support the scientific objectives and goals of ECMWF's 2025 strategy [7], both the resolution and the diversity of the generated data must increase substantially. From historical trends, one third of the growth has resulted from increases in resolution, and two thirds from an increase in diversity due to addition of more physical parameters, vertical levels or ensembles.

I/O performance growth has not kept pace with the available computational capacity. Significant software engineering and changes to the use pattern of HPC hardware will be required to support the data volumes of the future along with the increased metadata load implied by the increase of diversity. A number of new technologies are becoming, or shortly to become, available which will help with these challenges. ECMWF is already involved in projects that are looking to develop this capacity [20]. However, the distribution of these resources within an HPC environment is likely to be very different from the large-scale parallel filesystems that currently dominate. Not only will a software evolution be needed to support these various technologies, but it must be able to cope with differing distributions of storage capacity, and the potential lack of a global namespace in the manner of current parallel filesystems.

In addition to the scalability challenges caused by increasing data volumes, the number and diversity of consumers of data are also growing. Commercial customers and member states are increasingly wanting to access the full high-resolution output data set to perform their own computations, rather simply receiving a set of post-processed products from ECMWF. This trend is visible in the upcoming European Weather Cloud [27] which showcases the gradual trend to bring compute closer to the data as well as the continuing convergence of HPC and big data technologies.

A consequence of this is the need to provide significantly more access to the high-resolution model data within a cloud environment. ECMWF needs to increase not only the flexibility and capability of the output components of the model's I/O stack, but also the ability for processes running outside the HPC environment to access data within the FDB, and the configurational flexibility of both data storage and access. An overview of the overall computational infrastructure including potential future cloud services is given in Figure 1.

Previous work [26] has described the development of version 5 of the FDB, the development of backends that ensure appropriate data writing and access semantics, and its integration into the wider MARS ecosystem. The software described in that work has now entered routine use as part of the meteorological research workflows at ECMWF, and has entered the pipeline for operational use.

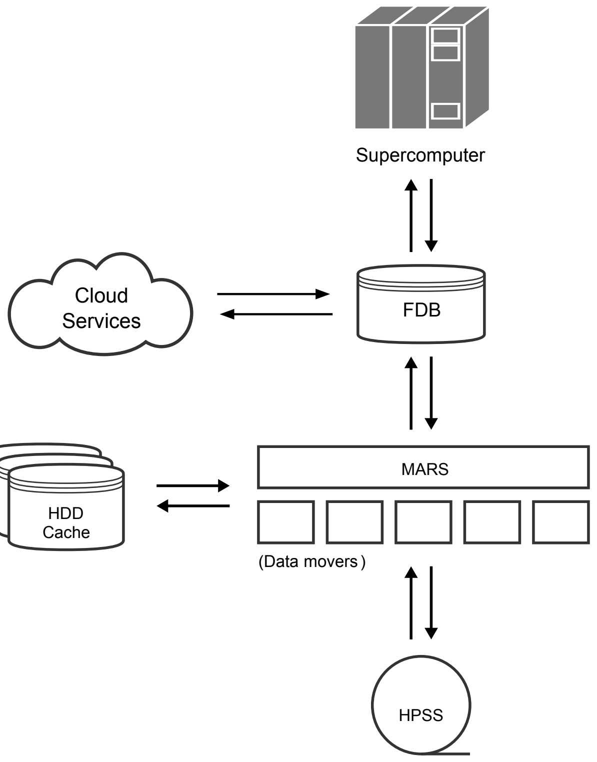


Figure 1: Overview of potential future high-performance data infrastructure at ECMWF.

In this paper we describe extensions to the FDB to provide a high-performance, distributed object store managed as a service. This has been achieved by abstracting the client API into a flexible runtime-configurable frontend giving fine grained control over *where* and *how* data will be stored. In addition to being able to forward data over the network, these frontends allow routing according to metadata, distribution using a Rendezvous hash [28], filtering and use of the FDB pluggable backends to persist data. A thin server has also been developed to which the remote frontend instances can forward the API requests and data.

1.1 Approaches to Data Storage

Development of ECMWF's I/O stack attempts to satisfy two different constituencies. Firstly, in common with most HPC applications, the needs of production forecasts demand high-performance data handling systems. This is matched by the needs to support our meteorological researchers in abstracting the technical details and questions of how and where data is to be handled. As a result, we consider it essential that the APIs and systems for data handling are *driven and controlled* through the use of domain-specific, scientifically meaningful metadata.

ECMWF considers management of data lifetimes to be key to its operational success. Systematically tagging and tracking the metadata associated with all generated meteorological data facilitates this process whether it is merely cached for seconds or archived for decades. It does, however, introduce complexity into the data handling pipeline.

Many weather and climate models make use of MPI-IO [30] to facilitate high-performance I/O for large volumes of data on parallel filesystems. This is often embedded into the HDF5 [29] library, and in turn into NetCDF [31] which provide a self-describing hierarchical containers for data with attached metadata, and may make use of such conventions as the Climate and Forecast (CF) metadata convention [6]. But this in itself does little to solve the data management and curation problem, and instead shifts this burden directly onto the application developers or the end-users. As the WMO has specified the GRIB format in detail, with a well-defined metadata schema, introducing HDF5 or NetCDF into the forecast pipeline increases rather than decreases the scale of our metadata handling challenges.

The FDB is a library and a data-management service that provides the storage layer for a multi-producer multi-consumer system. A significant number of processes within the operational workflow are responsible for generating, writing, and indexing meteorological data, and others access it. The FDB provides infrastructure such that this can all be done in a manner that is driven by metadata made up of dictionaries of scientifically meaningful key-value pairs. It provides an indexed data store for efficient access to the data with transactional and resilient semantics.

In the weather and climate field, the authors are not aware of other centres making use of indexed data stores within their data pipelines. Rather, the common practice is to focus on writing NetCDF or GRIB files to efficiently known locations in storage.

In contrast to typical HPC practice, where a large part of I/O usage is defensive in nature consisting of large blocks of checkpointing data, I/O usage in operational forecasts consists of very large numbers of independent fields. Accordingly, hardware parallelism can be exploited by interleaving data from multiple nodes according to a defined pattern, by task-parallelising the data writing across a set of I/O server nodes, or a combination of both. One approach is demonstrated by the XML I/O Server (XIOS) developed for weather and climate data [18]. In an approach with multiple writers systematically writing non-conflicting data, the guarantees provided by a POSIX filesystem hinder performance more than they assist the application.

ECMWF is by no means alone in observing the difficulties posed by POSIX semantics to extreme-scale I/O involving many readers and writers. The most obvious direction to relax POSIX semantics is a move towards object storage, and this points in a direction of very active research at different scales. There are numerous systems providing object storage in different contexts [1, 3, 4, 12, 16, 22]. These ultimately reduce to various forms of hash table, with data being stored or returned according to a supplied key.

Unfortunately, these hash-table related stores are not optimal for metadata-driven data flows. In particular, accesses using enumerated range requests and wildcards, or exploring the stored data, require a more extensive index which, for these general purpose object stores, would require an external indexing service. We consider that an approach to building a domain-specific data service with associated APIs is likely to be more productive[5], and as such he MARS ecosystem and the FDB are built around operations driven by field-related metadata and the metadata schema.

Beyond these more general purpose object stores, more specific development is being invested into HPC solutions. The SAGE [24] and SAGE2 [25] projects are working to explore the needs of extreme-data systems, and advanced object storage is being developed in conjunction with Seagate [23]. The DAOS project [14] at Intel aims to comprehensively replace the I/O stack with a system built around object storage, with a great deal of flexibility for application developers to make use of novel data movement models. ADIOS2[11] provides a flexible and performant framework to file and object I/O in scientific contexts.

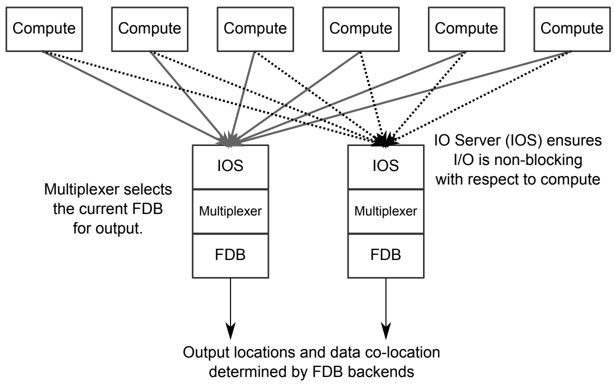

Compute
Compute
Compute
Compute
Compute
Compute
IO Server (IOS) ensures I/O is non-blocking with respect to compute
Multiplexer selects the current FDB for output.
IOS
Multiplexer
FDB
IOS
Multiplexer
FDB
Output locations and data co-location determined by FDB backends

Figure 2: The location of the Fields Database within the I/O output stack

Unfortunately, these systems are far from being in use in operational centres. The FDB developments at ECMWF have built a system that offers many of the upsides of other object storage systems, now, in a manner that is built around dedicated domain-specific metadata functionality. Backends built using other technologies can be integrated into this existing infrastructure as they become available. The work in this paper looks to extend the domain of this metadata-driven infrastructure to increase its flexibility and reduce the dependence on high-performance parallel filesystems.

1.2 The Fields Database (FDB)

The Fields Database (FDB) is both a software library and a service which have (in various forms) been in use for many years and have undergone substantial evolution. As a library, it provides the final output stage for the I/O stack used by the Integrated Forecasting System (the IFS, ECMWF's forecast model), as well as controlling meteorological data output for other tools. In current use it defines data structures and usage patterns to store meteorological data on a high-performance parallel filesystem, currently Lustre.

In this role, the FDB is just one of a number of components in the I/O stack. The role of the FDB, sitting at the bottom of the I/O stack is shown in Figure 2. Sitting above the FDB, the IFS's I/O server (IOS) aggregates distributed field data from where it is generated, distributed across the compute nodes in a simulation, into localised fields on one of a number of I/O servers. The simulation may then continue unhindered by time taken for I/O processes, and a parallel data output problem is turned into one with many independent but internally serial streams. A data multiplexer then gives runtime configurability to what I/O stack components are used, in this case selecting the current version of the FDB. The FDB does the actual I/O and is in control of *where* and *how* the data is written.

As a service, the FDB acts as a layer within the MARS ecosystem, storing recently generated meteorological data as an in-HPC hot cache of data. This data can be directly accessed using the FDB by other components within operational and research weather forecasting pipelines, but it is also served by instances of the MARS server software sitting on nodes within the HPC facility, to make data within this cache available efficiently to clients outside of the HPC facility. This is relevant as forecast data declines in value extremely rapidly with time. As a result the *vast* majority of data accesses are to recently produced data. In practice, roughly 80% of data requests to MARS are served from the FDB, with 80% of the remainder being retrieved from the disk cache in front of the tape library, such that only 4% of total requests reach tape.

In accordance with ECMWF's outlook on data handling and curation, all write and read operations, and data exploration are metadata driven, using scientifically meaningful metadata. Operations are both driven by and validated according to a well defined schema.

Development of the current version of the FDB (version 5) was documented in a paper presented at PASC in 2017 [26]. This development was driven by the need to more tightly integrate the FDB within the wider MARS ecosystem, to improve robustness by allowing multiple unrelated processes to write to the FDB simultaneously without explicit application-level synchronisation and provided defined behaviour on catastrophic error or under failure conditions, and to provide tightly defined consistency and transactional behaviour under write. The software developed in that paper has now been brought into use within the research pipeline at ECMWF, and is moving through the pipeline for rollout in the the daily operational forecasts. It is anticipated that version 4 of the FDB will have been entirely removed from the operational stack during 2019.

The work in this paper aims to extend the functionality of this object store and to decrease our coupling to specific hardware solutions. In particular, we aim to reduce our dependence on POSIX filesystems and any other globally-namespaced storage system. ECMWF also wishes to exploit the development of upcoming products and the output of various research projects [14, 17, 24]. We aim for a combination of *correctness*, *flexibility* and *performance*.

The FDB software will be released publicly in time for the completion of the NextGenIO project[20], including all new developments discussed in this paper. It will be found on ECMWF's github page[9].

2 DESIGN OF THE FDB

In existing operational use, the FDB library is able to act on globally visible parallel storage resources (currently a Lustre filesystem). As a result, it is able to exist solely as a library which provides an I/O layer to calling applications, managing how an application uses "local" resources. This library ensures that the use of these resources provides a precise semantic for data availability and consistency in the presence of many simultaneous reader and writer processes.

The FDB5 is an integral part of the MARS infrastructure. It needs to scale efficiently to an arbitrary number of writing processes whilst retaining a continuously consistent state from the perspective of an arbitrary number of reading processes, irrespective of any potential failures. The design of the FDB5 exploits knowledge of the particular write and read patterns in both the research and operational workflows at ECMWF.

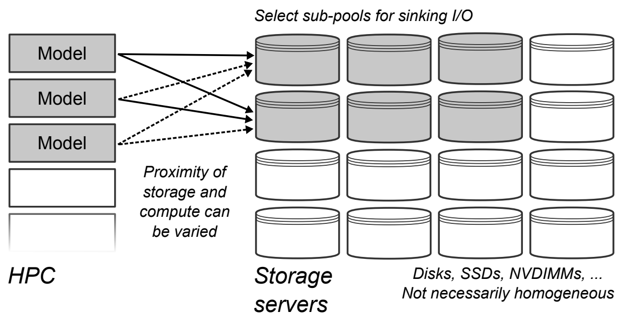

Select sub-pools for sinking I/O
Model
Model
Model
Proximity of storage and compute can be varied
HPC
Storage servers
Disks, SSDs, NVDIMMs, ... Not necessarily homogeneous

Figure 3: Selecting from a pool of FDB Servers

As the volume of data produced and consumed continues to grow, this poses extreme requirements on the capacity of the parallel filesystem — especially in terms of metadata handling. We can neither assume that the performance of globally-namespaced parallel filesystems will scale with our data needs, nor that they would be financially sensible propositions if they did.

Previous development work on the FDB has focussed on backends that describe *how* the data is to be stored. These developments have provided ECMWF with a system with the correct semantic, and that is sufficiently performant to handle our data needs in the immediate future. The projected growth of data use in the future suggests that the current and future focus needs to include *where* data is to be stored. The work in this paper provides a framework under which the data location questions can be approached.

One possible usage pattern is to have a pool of (relatively) commodity servers with locally attached persistent storage, and balanced networking connectivity. According to curated configuration, components of the operational model or of research experiments will, in a deterministic fashion, select a subset of this pool of servers to use for storing data. The data can then be retrieved from these servers. For each field, these servers can then be visited according to a Rendezvous Hash to distribute the data [28]. An example of this usage is illustrated in Figure 3. This setup allows us to use different amounts of hardware of significantly varying performance (and thus cost) for different workflows according to their needs and importance.

This has the side effect of potentially making the data in the FDB directly visible from machines outside of the HPC facility. This facilitates a number of enhancements to our workflows that are highly desirable:

- We will be able to directly transfer data between the FDB and other machines in the MARS ecosystem, without batching a job on the HPC machine, and reading all the data to a compute node before re-archiving it.
- Users will be able to retrieve data directly from the FDB to machines on our normal network, without having to proxy data through an intermediate server, which becomes a bottleneck.

- The data is more straightforwardly made available to a in-premises cloud infrastructure located close to the HPC system.

However, implementing a runtime-configurable system requires a great deal more flexibility than is available in the existing backend architecture of the FDB. The following sections describe how the FDB has been reconfigured to include both a configurable backend and a composable frontend architecture. The frontends provide the functionality to route data according to the metadata, distribute the data across available resources and connect to remote storage resources, thus providing horizontal scalability. The API frontends need to do this whilst preserving the precisely controlled semantics provided by the existing backend functionality.

2.1 Metadata handling

The FDB is a library that provides functionality to a range of scientific users. A primary goal is that development of meteorological application code, and use of the data by users, require no knowledge of where and how the data is to be actually stored — indeed this can be changed by the operations team without involvement of the meteorologists. As a consequence, *all* of the API calls are driven by metadata in the form of a dictionary of key-value pairs of scientifically meaningful description.

As an example, the MARS client allows a user to retrieve data (and, if that data is sufficiently recent, will do so from the FDB). An illustrative example of a MARS request is shown in Listing 1. This listing selects data from one member (`number=13`) of an operational (`class=od,expver=0001`) ensemble forecast (`stream=enfo,type =pf`), from a specified base forecast time (`date=20190109,time =1200`) for a specific set of forecast times (`step=1/to/240/by/3`) selecting data for temperature and tangential components of wind on all model levels (`param=t/u/v,levelist=1/to/90`).

Listing 1: A MARS request to retrieve operational ensemble forecast data that may be in the FDB

```
retrieve,
    class = od,
    type = pf,
    stream = enfo,
    expver = 0001,
    levelist = 1/to/90,
    param = t/u/v,
    date = 20190109,
    time = 1200,
    step = 1/to/240/by/3,
    number = 13,
    target = "output.grib"
```

This metadata handling is even more natural when considering the nature of the data being stored. The majority of the data stored in the FDB is in GRIB format [33]. This format is self-describing, and the key-value dictionary for uniquely identifying the data can be extracted from the encoded GRIB data. When archiving data, the FDB uses this fact to check that the data being archived matches the data that the calling application says it is archiving.

The metadata and the MARS language follow a data model which is externally managed and versioned as part of data curation and stewardship. Within the FDB this is enforced by a schema. A versioned copy of this is stored alongside the data indexes to ensure old data is always accessible.

Enforcing that all data is generated with associated metadata matching a curated schema greatly facilitates long-term data management and archiving. The FDB's metadata handling functionality is built out of the same building blocks as are present in other components of the MARS ecosystem, and the FDB is effectively integrated as one layer forming an application-controlled hierarchical storage system. This ecosystem is able to find, enumerate, explore and describe *all* of the meteorological data that has *ever* been archived to it. Whilst this has come with up-front data modelling costs, the long-term benefits are enormous.

2.2 Front- and backends

The FDB has pluggable backends [26] that define *how* data and metadata are stored and made available given the hardware available. The backend is responsible for ensuring that the semantics of data archiving, listing and retrieval are provided on top of whatever provisions are made by the underlying system. In particular:

(1) Data is either visible, and correctly indexed, or not. The FDB behaves in an ACID [13] sense.
(2) `archive()` blocks until the FDB has taken control of (a copy) of the data. Data is not necessarily visible at this point.
(3) `flush()` blocks until all data from the current process that has been `archive()`ed is persisted into the underlying storage medium, correctly indexed and has been made visible to any reading processes.
(4) Once data is is made visible, it is immutable.
(5) Data can be replaced by `archive()`ing a new piece of data with the same metadata. This second archive shares the semantics of the first, such that the *old* data is visible until the new data is fully persisted and indexed.

In use at ECMWF, data in the FDB is visible to all users with access to the HPC facility. The FDB backend is responsible for ensuring that data may only be overwritten or supplemented by the user who created it.

When the FDB was being used purely on top of high-performance, globally-visible storage systems, having pluggable backends sufficed for configuration flexibility. However, it is anticipated that upcoming heterogeneous storage systems will not necessarily have globally visible storage components. Further, from a configuration perspective, it is desirable to have more flexibility for the data curation team to modify read and write behaviour for different types of data separately.

The API being presented by the FDB to calling applications was extracted into a concise abstract API, a subset of which is given in Listing 2. At runtime, this API is fulfilled by composable frontend instances, constructed by a factory which makes use of global, local or application-defined configuration. These instances are built from a range of specified types, each of which must implement all of the the functionality in the API. Each of the instances is intended to be very thin and to either a) forward API calls to further instance(s), or (b) contain an FDB backend that receives the API calls.

Listing 2: Extract of FDB API

```
class FDB {
public: // ------ Primary API functions ------

    void archive(const Key& key,
                 const void* data,
                 size_t length);

    void flush();

    DataHandle* retrieve(const MarsRequest& request);

    ListIterator list(const FDBRequest& request);

    DumpIterator dump(const FDBRequest& request);

    WhereIterator where(const FDBRequest& request);

    WipeIterator wipe(const FDBRequest& request);

    PurgeIterator purge(const FDBRequest& request);

    StatsIterator stats(const FDBRequest& request);
```

These composable frontend instances can introduce a range of behaviour including asynchronicity, threading, networking, data mirroring and filtering to the I/O pathway with no changes being made to the calling applications. This fully extracts responsibility for data storage decisions (particularly data locality) to the operations team allowing the meteorologists to focus on science.

2.2.1 Exploring existing data. The majority of API calls involve an exploration of the stored metadata. In the backends this is implemented by means of a visitor pattern. Each API call corresponds to a visitor (a function object which stores its own data and is executed once for each element which is visited). The backend provides a mechanism to walk the metadata schema in conjunction with a supplied request to find all matching data elements. In turn, the visitor generates a sequence of output elements. As an example, the visitor corresponding to `list()` generates `ListElement` objects which contain a metadata key along with a location descriptor for the data. By contrast, the visitor corresponding to `stats()` generates one `StatsElement` object per metadata index that is encountered (and these elements can be aggregated to provide overall statistics).

To provide an iterable API, the local frontend instance invokes the backend mechanism within a thread. The visitors then push output elements onto a queue. The returned Iterator object can then pop output elements from this queue as driven by client code. This inverts the control, such that the iterator objects do not have to maintain all state required to perform a "next" operation for exploring the metadata. The queues can be configured for a maximum size and memory usage, and when they are full this blocks the visitor as it pushes the next element onto the queue, throttling the rate, and allowing for an enumeration of an arbitrary size within a fixed memory constraint.

2.2.2 Writing Data. Archival is a two step process, but semantically the API is very straightforward. Calls to `archive()` contain a data blob with associated metadata, and these are directly forwarded on to the appropriate backend which blocks until it has taken any necessary copies of the data.

The semantics of the FDB are that a workflow cannot assume that data is visible to reading processes until `flush()` is called, and that this data is potentially at risk of loss in case of failures. This means that `flush()` will block until all data that has been `archive()`d is persisted, indexed and visible. The frontend simply forwards the the call to the appropriate backend(s) which are 'dirty' (i.e. have received data).

2.2.3 Retrieving Data. Data retrieval is slightly more complex, as we do not want to assume that all requested data can fit into memory at any point, as clients may easily request large data sets, say 200-400 GiB in a single request. The `retrieve()` API call returns an object of type `DataHandle`. This contains sufficient location information for the client code to directly retrieve the data. For a request that retrieves more than one field, a number of these `DataHandle` objects can be aggregated into an aggregate `DataHandle` object which will retrieve the fields sequentially. The results of a number of `retrieve()` calls can be similarly combined.

As a result, frontend instances forward the request to the required backend(s) which block whilst walking the index to find the data. The DataHandle is then returned, and the data can be retrieved or streamed at a later point.

2.3 Sample Frontends

When the FDB is configured, the API is provided by concrete frontend instances from one of a number of specified types. These may include functionality for:

Routing Perform pattern matching on the key/value pairs in the associated metadata, and use this to select which sub-frontend instance is used.
Distribution Deterministically distribute API calls to a series of sub-frontends according to a Rendezvous Hash [28] of the associated metadata. This can be used for load-balancing between multiple storage resources.
Mirroring Duplicate API calls to multiple sub-frontends. If non high-availability storage resources are being used, this mechanism can provide redundancy.
Diagnostics Forward API calls to another instance while performing diagnostics and logging on the API call traffic.
Filtering Select API calls to forward or to discard data according to some criterion. This enables configuring the output behaviour of a forecast without changing the code or runtime configuration (and code paths) of the forecast *per se.*
Remote forwarding Forward API calls to an FDB server on another host. See section 2.4.
Local Interact directly with a storage backend.

Figure 4 demonstrates a sample configuration for model output corresponding to the YAML [2] configuration in Listing 3. A written field is first passed to a `routing` instance which determines that the field is an operational output field (as opposed to, say, research data).

Listing 3: Example API frontend configuration

```
type: routing
fdbs:
- type: dist
```

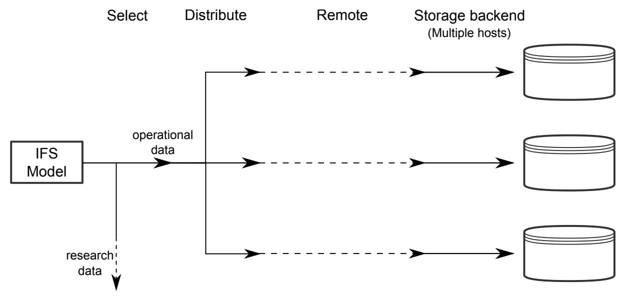


Figure 4: A sample frontend configuration

```
select: class=od
lanes:
- type: remote
  host: host-one
  port: 7656
- type: remote
  host: host-two
  port: 7656
- type: remote
  host: host-three
  port: 7656
```

2.4 FDB Server and Remote Storage

A new component has been developed called the FDB server. This is a thin server which can be run on storage nodes that have their own local persistent storage, and are remote from the client code. Each instance of the FDB server has its own, local, FDB configuration which instructs it how to make use of its local storage.

A client remote FDB frontend instance is configured with a host and a port (as in Listing 3). At runtime, each of these clients opens a control connection to the thin server, and then a data connection on a port supplied by that server. The server maintains one process for handling each of the connected clients.

In current design the FDB is intended for use only within a trusted network environment. As such it implements no authentication mechanisms. As the clients open control connections to the server only once, and maintain it for the life of the application, authentication could be straightforwardly added at the time the connection is established with minimal performance overhead.

The client and server communicate by means of messages in a standardised form. A fixed size header contains an identifying magic value, a message ID, a tag identifying the message type, and the size of any payload. The payload then follows as raw bytes, followed by a tail marker which can be used to check that the message is correctly received. The message ID is unique per client process per API request, and is included in all response and data messages resultant from that request.

Messages on the control connection will always be expected, and are handled immediately. Messages on the data connection are always received into a listening thread, before being pushed onto a thread-safe queue. Messages are then pulled out of that queue by a handling thread. This decouples the communication from the handling of messages and allows the overall process to be limited by the slower of the network or backend API, rather than repeatedly blocking on both.

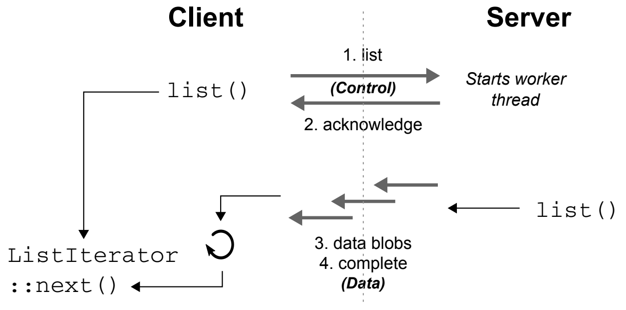


Figure 5: Listing data stored on a remote FDB

The FDB server maintains a listening thread on the control connection at all times. It starts worker threads as required that listen or write back to the data connection. Once handled, all messages it receives are responded to with an `ACK` (acknowledge) message (which is expected by the client). Only one process that receives messages on the data connection may exist at any one time on the server (corresponding to ongoing `archive()` operations).

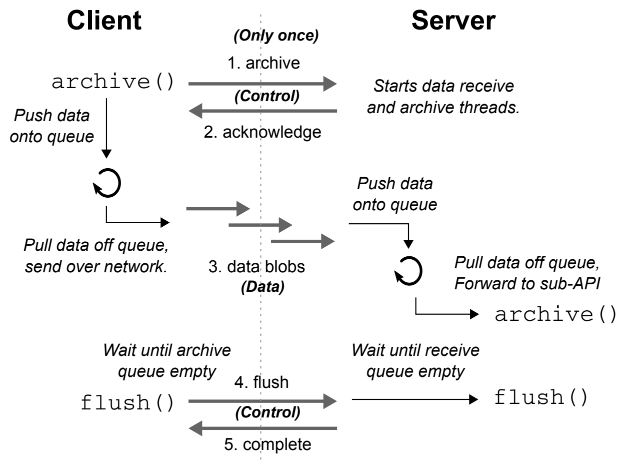
A listening thread will exist whenever messages may be received on the data connection on the client. It is legitimate to have multiple ongoing requests simultaneously if supported by the backends, so a thread-safe queue is maintained per ongoing request ID. Messages are pushed onto the correct queue according to the request ID in their header. Messages on the data connection corresponding to `retrieve()` operations are handled slightly differently as described in Section 2.4.3.

2.4.1 Exploring existing data. Most of the API functions operate in the same way. The constraint is that the data elements returned must be serialisable using the serialisation components of ECMWF's eckit library [8]. As the structure of how these calls are forwarded across the network is identical, the system is vastly simplified by sharing much of their implementation.

We will take the `list()` function as an example, and an overview of the process can be seen in Figure 5. The first thing that the client side process does is ensure that a thread-safe queue exists for a newly acquired request ID, and then starts a worker thread listening to this message queue. Then the request is serialised into a memory buffer. Along with a tag identifying this as a `LIST` operation and the request ID, the request is sent as a message to the server across the control connection. The control server will start a worker thread before responding with an `ACK` message. On receipt of this `ACK`, the client returns an asynchronous iterator (containing an internal thread-safe queue) to the calling code.

On the server side, the worker thread calls the `list()` function on the locally configured frontend instance, and iterates on the `ListElement` objects accessible from the returned `ListIterator`. Each of these `ListElements` are serialised into a buffer, and written to the data socket back to the client as part of a message with the corresponding request ID and message type `BLOB`.

This message is received by the listening thread loop on the client side, which reads the message ID and immediately deposits this message in the relevant queue before returning to listening on the



Client
Server
(Only once)
1. archive
archive()
Starts data receive and archive threads.
(Control)
2. acknowledge
Push data onto queue
Pull data off queue, send over network.
3. data blobs
(Data)
Push data onto queue
Pull data off queue, Forward to sub-API
archive()
Wait until archive queue empty
Wait until receive queue empty
4. flush
flush()
(Control)
flush()
5. complete

Figure 6: Archiving and flushing a sequence of fields to a remote FDB

data socket. The worker thread decodes the message, deserialises the `ListElement` object and pushes it onto the queue associated with the output iterator where it can be retrieved by the calling code.

The overall rate of movement of the `ListElement` objects from the server to the client may be limited by the communication over the network. Alternatively, if the client is retrieving `ListElements` slower than the network supplies them, then the queue in the output iterator will become full, followed by the message queue corresponding to the request ID. Once the queues are full, the next push operation will stall until a `ListElement` is retrieved downstream. No more data will be read from the socket, and the server will wait on a write operation.

On the server side, once all messages have been read then a `COMPLETE` message is written to the data socket with the appropriate request ID, and the worker thread exits. On the client side, the listening thread will then mark the appropriate queue as completed, and this will propagate up the chain of worker threads and queues to the calling code.

2.4.2 Writing Data. Writing data is probably the most complex part of the API, and operates under different performance and semantic constraints. In particular, we wish to minimise the time taken inside `archive()` API calls on the client side (as these directly slow down the calling code). We also wish to minimise the duration that the code must block for when `flush()` is called.

As a result, we take two decisions:

(1) Only two blocking calls will be made on the control connection. One for the first `archive()` call and one for the `flush()` call.
(2) We eliminate network chit-chat (and thus time spent waiting on acknowledgement of responses) by writing messages containing the fields to the data connection sequentially as fast as possible, without waiting for acknowledgements from the server side until the `flush()` command.

The overall process of writing data to the server is illustrated in Figure 6. When the first `archive()` call is made, an `ARCHIVE` message is sent to the server. The server starts two worker threads.

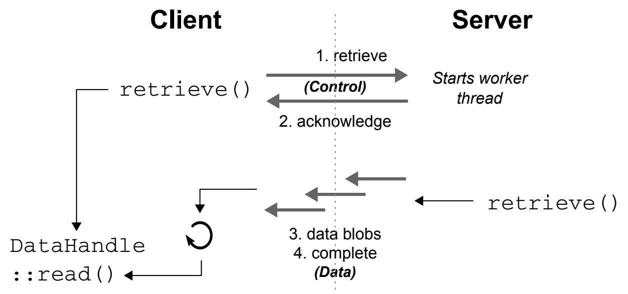


Figure 7: Retrieving data stored on a remote FDB

One of these listens on the data socket on the server side, receiving a sequence of data messages (of type `BLOB`) and pushes these onto a thread-safe queue. The second thread pops these messages from the queue, extracts the archiving metadata and the field from the message and calls the `archive()` function on the configured frontend instance.

Once these threads have been started, the server sends an `ACK` (acknowledge) message back to the client side, which in turn spawns an archiving thread. This thread reads from a queue, encodes metadata and a field into a data message of type `BLOB` and writes them to the data socket.

As only one `ARCHIVE` message is sent to the server prior to `FLUSH`, there will only ever be one listening worker thread on the server side. Further calls to the `archive()` API method will simply push a buffer containing the field data, and a copy of the metadata, onto the archiving queue, from where it will be shuttled over the network connection and passed to the downstream frontend instance while being buffered and serialised over the network as appropriate.

Once all the relevant calls to `archive()` have been made, the client code will call `flush()`. This triggers two processes. Firstly, the archive queue is marked as closed. When this queue is then emptied, the archiving worker thread will send a `FLUSH` message across the data connection. The appropriate listening thread on the server side will then mark its receive queue as closed before exiting. The final archiving thread will complete and exit once it has called `archive()` with all the received data.

Secondly, a `FLUSH` message will be sent across the control connection. On the server side, the handler will block until the archiving worker threads have completed, then call `flush()` on the configured frontend instance before responding `ACK` to the client.

2.4.3 Retrieving Data. In retrieving data we must be careful never to require all of the requested data to be cached in memory at any one time. For a large request, this can easily overflow the memory available for the process on either the client or the server machine. We must generate a multi-stage pipeline such that reading from the returned `DataHandle` object results in data being read by the backend as required, and essentially streamed to the calling client.

On a per-request basis, this process is very similar to the one for API calls that explore data, as presented above, and is illustrated in Figure 7. However, the additional constraint is that a typical use pattern for the `retrieve()` call is to make a large number of calls, and aggregate the returned `DataHandle` instances into a container `DataHandle` and then retrieve all of the data lazily.

Client	Aggregate			Server per-process
Processes	Fields	Data	MiB / s	MiB / s
1	12 600	38.51	187.23	295.13
2	25 200	77.02	394.84	392.66
4	50 400	154.03	615.90	220.70
8	100 800	308.06	708.10	75.81
16	201 600	616.13	1080.22	43.73
32	403 200	1232.25	1037.89	22.46
64	806 400	2464.51	989.80	9.81
128	806 400	2464.51	923.34	5.07
256	752 640	2300.21	914.86	2.66

Table 1: Initial throughput tests with 32 client nodes and two storage servers fitted with spinning disks

Client	Aggregate			Server per-process
Processes	Fields	Data	MiB / s	MiB / s
1	12 600	38.51	178.82	556.48
2	25 200	77.02	353.46	505.27
4	50 400	154.03	665.55	493.11
8	100 800	308.06	664.37	528.82
16	201 600	616.13	681.76	549.71
32	403 200	1232.25	570.87	558.43
64	806 400	2464.51	640.27	561.34
128	806 400	2464.51	616.24	557.43
256	752 640	2300.21	589.85	549.81

Table 2: Initial throughput tests with 32 client nodes and four storage servers with NVMe SSDs

As such, we introduce a queue of `DataHandle` objects on the server side through which all of the retrieve requests are handled. Further, rather than maintaining a queue of received messages per request ID, all data messages received are placed in one queue. This maintains the queue sizes and essentially combines all of the requests into one stream under the constraint that the data is read in the order it is retrieved.

The client side maintains a message queue for data messages corresponding to retrieve operations (with a configurable maximum size), and the continuously running listening loop will push messages of type `BLOB` or `COMPLETE` onto this queue.

Each `retrieve()` API call results in a blocking exchange over the control connection. Once the client has received the `ACK` message from the server, it returns an object of type `DataHandle` which contains the request ID and a reference to the received message queue.

On the server side, once a `RETRIEVE` message has been received it is immediately forwarded to the `retrieve()` method on the configured frontend. This returns a `DataHandle` instance which is pushed onto a thread-safe queue, before responding with an `ACK` message to the client.

A worker thread pops the `DataHandle` instances off the queue in sequence, one at a time. The data associated with each `DataHandle` is then read in chunks (typically 10 MiB) and written as messages, along with the corresponding request ID, to the data socket back to the client. Once each `DataHandle` is exhausted, a `COMPLETE` message is sent to the client with the same request ID.

As the data is read by the client, the data handles pop the messages off the received queue and return the contained data. For this queue mechanism to work, the `DataHandle` objects returned to the client must be read in the order they are requested.

3 USAGE AND RESULTS

3.1 Small Test Clusters

ECMWF has done a significant amount of robustness testing of the distributed and remote FDB functionality, and a small amount of performance testing. We have made use of our small testing compute cluster on which we have been able to run dedicated loads up to 32 nodes with no external interference.

The first tests were carried out against two servers used for testing MARS code developments, each with local spinning-disk based RAID arrays. Initial results are presented in Table 1. As the level of concurrency builds, the aggregate rate increases to just over 1 GiB / s. It is notable that the rate seen per server process, between the server instances and the spinning disk, declines sharply as the number of concurrent writers is increased, whilst the aggregate rate stays roughly constant. This indicates that with these spinning disks on a small number of servers we are limited by the performance of the disks.

We have access to a small testing platform of four nodes that have been fitted with higher performance storage systems, two with 700 GiB NVMe NAND SSD devices and two with 700 GiB 3D XPoint SSDs. However, as seen in Table 2, as concurrency was increased it is clear that using these nodes the throughput was tailing off at only around 650 MiB / s. It turns out that the network connectivity between our testing compute cluster and this experimental storage cluster was substantially poorer than for the other systems, and the results observed demonstrate the network throughput. It is worth noting that the throughput to the SSDs observed by the server processes is almost constant through these tests indicating that we were a long way from saturating the storage devices.

Ultimately, these tests have been limited to demonstrating the robustness of the data semantics provided by the API frontends and the FDB server under the presence of hundreds of writing processes and artificially induced failure conditions.

3.2 NextGenIO Prototype

The NextGenIO project [20] is a research project funded by the European Union which aims to develop a hardware platform with new technology bridging the gap between memory and storage. As part of this project, a novel architecture for HPC nodes has been developed [19], including a large amount of Storage Class memory (Intel® Optane™ Data Center Persistent Memory DIMMs [15]), distributed across the compute nodes, and an Intel® Omni-Path Architecture high-performance fabric.

At the time of writing, an initial test cluster containing four nodes of prototype hardware has been made available for preliminary testing. Although there are very few nodes available, it has been possible to obtain some initial performance data.

A test configuration was used with 12 processes of a benchmarking tool which imitates the I/O writing behaviour of ECMWF's forecast model running on each of the nodes. These used the FDB configured to distribute data across all four of the nodes using Rendezvous Hashing and the high-performance fabric, where it was persisted to the storage class memory.

The aggregate performance using these four nodes was extremely good, achieving a sustained end-to-end aggregate of approximately 8 GiB / s of application-generated data (including all correct metadata handling, checking and indexing). Scaled up to the size of our operational storage systems, this would imply significant performance improvements on less specialised (and thus less expensive) hardware.

A larger scale (34 node) prototype cluster will become available by February 2019 on which more meaningful performance testing can be carried out at a scale which more closely matches the I/O configuration of actual operational use. Meaningful performance numbers should be available in time for the PASC conference in June 2019.

4 CONCLUSIONS

This paper has presented developments to the FDB object-store used in the data pipeline for ECMWF's research and operational NWP workloads. These changes enable the FDB to operate as a distributed object-store for meteorological data, with data stored on a selection of storage servers each with local storage.

An API provided by runtime-configurable frontends has been introduced, and this can be configured with various concrete composable frontend instances. The frontend types already implemented include functionality to distributed data and API calls using Rendezvous Hashing, and the ability to forward API calls over the network. A thin FDB server has been implemented to provide the managed storage service used by the clients.

The frontends provide a great deal of flexibility as to *where* data is to be stored. This is in addition to the control of *how* the data is to be stored and indexed that already existed as a result of previous work.

These developments have been stress tested for correctness and flexibility, and appear to be giving performance which is largely limited by network capacity in the cases that have been tested. However, serious performance testing is dependent on the availability of upcoming hardware, and is likely only to be undertaken by end of Q1/2019.

The additional flexibility and configurational options made available by the new developments are anticipated to play a significant role in ECMWF's ability to continue to support its NWP workloads in the face of exponentially increasing data volumes and the needs of cloud computing. It is anticipated that the additional flexibility will significantly decrease both cost and risk involved in provisioning future storage resources. We look forward to integrating these changes into our research and operational pipeline as part of the migration of our data centre to a new location in 2020.

Beyond this work, ECMWF is also participating in a range of further projects to understand and explore the use and handling of data in large-scale workflows [10, 17, 20] and their interaction with the FDB object store.

ACKNOWLEDGMENTS

This work has been funded by the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 671951.

REFERENCES

[1] Amazon. 2016. AWS Storage Services Whitepaper. https://d0.awsstatic.com/whitepapers/Storage/AWS%20Storage%20Services%20Whitepaper-v9.pdf.
[2] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. 2009. YAML Ain't Markup Language Version 1.2. https://yaml.org/spec/1.2/spec.html.
[3] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7 (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1267308.1267323
[4] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *IN PROC. SOSP*. 205–220.
[5] Matthieu Dorier, Philip Carns, Kevin Harms, Rob Latham, Robert Ross, Shane Snyder, Justin Wozniak, Samuel Gutierrez, Bob Robey, Brad Settlemyer, Galen Shipman, Jerome Soumagne, James Kowalkowski, Marc Paterno, and Saba Sehrish. 2018. Methodology for the Rapid Development of Scalable HPC Data Services. 76–87. https://doi.org/10.1109/PDSW-DISCS.2018.00013
[6] Brian Eaton, Jonathan Gregory, Bob Drach, Karl Taylor, Steve Hankin, John Caron, Rich Signell, Phil Bentley, Greg Rappa, Heinke Höck, Alison Pamment, and Martin Juckes. 2011. NetCDF Climate and Forecast (CF) Metadata Conventions. http://cfconventions.org/cf-conventions/v1.6.0/cf-conventions.pdf.
[7] ECMWF. 2015. ECMWF Strategy 2016-2015, The strength of a common goal. http://www.ecmwf.int/sites/default/files/ECMWF_Strategy_2016-2025.pdf.
[8] ECMWF. 2019. eckit. https://github.com/ecmwf/eckit.
[9] ECMWF. 2019. ECMWF github. https://github.com/ecmwf.
[10] European Commission. 2019. Large-scale EXecution for Industry and Society. https://cordis.europa.eu/project/rcn/219768/factsheet/en.
[11] Exascale Computing Project, US DOE. 2018. ADIOS2. https://adios2.readthedocs.io/en/latest/.
[12] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894
[13] Theo Haerder and Andreas Reuter. 1983. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (Dec. 1983), 287–317. https://doi.org/10.1145/289.291
[14] Intel. 2014. DAOS API Design Document. https://wiki.hpdd.intel.com/download/attachments/12127153/DAOS%202.5%20DAOS%20API%20and%20DAOS%20POSIX%20Design%20Document.pdf.
[15] Intel. 2018. Intel® Optane™ Technology — Disruptive Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology-animation.html.
[16] Avinash Lakshman and Prashant Malik. 2010. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.* 44, 2 (April 2010), 35–40. https://doi.org/10.1145/1773912.1773922
[17] MAESTRO. 2018. MAESTRO website. http://www.maestro-data.eu/.
[18] Yann Meurdesoif, Ozdoba H., Caubel A., and Marti O. 2012. XIOS. http://forge.ipsl.jussieu.fr/ioserver/raw-attachment/wiki/WikiStart/XIOS_IO_Workshop_Hamburg.pdf.
[19] NextGenIO. 2018. Hardware Architecture, NEXTGenIO Architecture White Paper. http://www.nextgenio.eu/sites/default/files/documents/publications/NEXTGenIO_whitepaper2_OCT18_LO.pdf.
[20] NextGenIO. 2018. NextGenIO website. http://www.nextgenio.eu/.
[21] B. Raoult. 2012. Architecture of the new MARS server.
[22] Redis MPI Forum. 2017. Redis Cluster Specification. https://redis.io/topics/cluster-spec.
[23] SAGE. 2016. Data Storage for Extreme Scale; The SAGE Project Technical White Paper. http://sagestorage.eu/sites/default/files/Sage%20White%20Paper%20v1.0.pdf.
[24] SAGE. 2016. Percipient StorAGe for Exascale. http://www.sagestorage.eu.
[25] SAGE. 2016. Percipient StorAGe for Exascale Data Centric Computing 2. http://www.sagestorage.eu/content/sage2-overview.
[26] Simon D. Smart, Tiago Quintino, and Baudouin Raoult. 2017. A Scalable Object Store for Meteorological and Climate Data. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC '17)*. ACM, New York, NY, USA, Article 13, 8 pages. https://doi.org/10.1145/3093172.3093238
[27] ECMWF Stephan Siemen. 2018. Welcome:Workshop on Developing Python frameworks for earth system sciences. https://www.ecmwf.int/sites/default/files/elibrary/2018/18718-one-year-where-are-we.pdf.

[28] David Thaler and Chinya Ravishankar. 1996. A Name-Based Mapping Scheme for Rendezvous. http://www.eecs.umich.edu/techreports/cse/96/CSE-TR-316-96.pdf.

[29] The HDF Group. 2017. Parallel HDF5. https://support.hdfgroup.org/HDF5/PHDF5.

[30] The MPI Forum. 2012. MPI: A Message Passing Interface. http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[31] University Corporation for Atmospheric Research. 2016. NetCDF Format. www.unidata.ucar.edu/software/netcdf/docs.

[32] Richard W Watson and Robert A Coyne. 1995. The parallel I/O architecture of the high-performance storage system (HPSS). In *Mass Storage Systems, 1995.'Storage-At the Forefront of Information Infrastructures', Proceedings of the Fourteenth IEEE Symposium on.* IEEE, 27–44.

[33] World Meteorological Organization. 2015. GRIB Format. http://www.wmo.int/pages/prog/www/WMOCodes/WMO306_vI2/Publications/2015editionUP2018/WMO306_vI2_en_ONLINE.pdf.